Notes by Ben Zhao

1 Introduction to Compression

- Information Theory
- Lossless Compression
  1. Huffman Coding
  2. Shannon-Fano Coding
  3. Arithmetic Coding
  4. LZ78 and LZW (GIF, UNIX Compress...)
  5. LZ77 (gzip, ZIP, ...)
  6. PPM (prediction by partial matching)
- Lossy Compression
  1. Transform Coding (Blocks: JPEG, MPEG, Wavelets...)
  2. Vector Quantization

1.1 Information Theory

How much can we compress any given data? It Depends on how much "information" the data contains.

Entropy

Shannon borrowed the the notion of *entropy* from statistical physics to capture the information contained in an information source. For a set of possible messages S, it is defined as,

$$H(S) = \sum_i p_i \log_2 \frac{1}{p_i}$$

where p_i is the probability of i^{th} message that can be sent by the source. The entropy is a measure of the average number of bits needed to send a message, therefore larger entropies represent more information. Perhaps counter intuitively, the more random a set of messages (the more even the probabilities) the more information they contain. The terms $\log_2 \frac{1}{p_i}$ represent the number of bits used to code message i (the information contained in S_i). Therefore messages with higher probability will use fewer bits.

	bits/char
bits $\lceil \log(96) \rceil$	7
entropy	4.5
Huffman Code (avg.)	4.7
Entropy (Groups of 8)	2.4
Asymptotically approaches:	1.3
Compress	3.7
Gzip	2.7
BOA	2.0

Table 1: Information Content of the English Language

Examples:

$$
\begin{aligned}
p_i &= \{0.25, 0.25, 0.25, 0.125, 0.125\} \\
H &= 3 \times 0.25 \times \log_2 4 + 2 \times 0.125 \times \log_2 8 \\
&= 1.5 + 0.75 \\
&= 2.25
\end{aligned}
$$

$$
\begin{aligned}
p_i &= \{0.5, 0.125, 0.125, 0.125, 0.125\} \\
H &= 0.5 \times \log_2 2 + 4 \times 0.125 \times \log_2 8 \\
&= 0.5 + 1.5 \\
&= 2
\end{aligned}
$$

$$
\begin{aligned}
p_i &= \{0.75, 0.0625, 0.0625, 0.0625, 0.0625\} \\
H & \quad = 0.75 \times \log_2(\frac{4}{3}) + 4 \times 0.0625 \times \log_2 16 \\
&= 0.3 + 1 \\
&= 1.3
\end{aligned}
$$

Note that the more uneven the distribution, the lower the Entropy.

1.2 The English Language

We might be interested in how much information the English Language contains. This could be used as a bound on how much we can compress English, and could also allows us to compare the density (information content) of different languages.

One way to measure the information content is in terms of the average number of bits per character. Table 1 shows a few ways to measure the information of English in terms of bits-per-character. If we assume equal probabilities for all characters, a separate code for each character, and that there are 96 printable characters (the number on a standard keyboard) then each character would take $\lceil \log 96 \rceil = 7$ bits. The entropy assuming even probabilities is

Date	bpc	scheme	authors
May 1977	3.94	LZ77	Ziv, Lempel
1984	3.32	LZMW	Miller and Wegman
1987	3.30	LZH	Brent
1987	3.24	MTF	Moffat
1987	3.18	LZB	Bell
.	2.71	GZIP	.
1988	2.48	PPMC	Moffat
.	2.47	SAKDC	Williams
.	2.47	PPMD	Howard
Nov 1993	2.34	PPMC	Moffat
Oct 1994	2.34	PPM*	Cleary, Teahan, Witten
18 Nov 1994	2.33	PPMD	Moffat
1995	2.29	BW	Burrows, Wheeler
31 Jan 1995	2.27	PPMD	Teahan
1997	1.99	BOA	

Table 2: Lossless compression ratios for text compression on Calgary Corpus

$\log 96 = 6.6$ bits/char. If we give the characters a probability distribution (based on a corpus of English text) the entropy is reduced to about 4.5 bits/char. If we assume a separate code for each character (for which the Huffman code is optimal) the number is slightly larger 4.7 bits/char.

Note that so far we have not taken any advantage of relationships among adjacent or nearby characters. If you break text into blocks of 8 characters, measure the entropy of those blocks (based on measuring their frequency in an English corpus) you get an entropy of about 19 bits. When we divide this by the fact we are coding 8 characters at a time, the entropy (bits) per character is 2.4. If we group larger and larger blocks people have estimated that the entropy would approach 1.3 (or lower). It is impossible to actually measure this because there are too many possible strings to run statistics on, and no corpus large enough.

This value 1.3 bits/char is an estimate of the information content of the English language. Assuming it is approximately correct, this bounds how much we can expect to compress English text if we want lossless compression. Table 1 also shows the compression rate of various compressors. All these, however, are general purpose and not designed specifically for the English language. The last one, BOA, is the current state-of-the-art for general-purpose compressors. To reach the 1.3 bits/char the compressor would surely have to "know" about English grammar, standard idioms, etc..

A more complete set of compression ratios for the Calgary corpus for a variety of compressors is shown in Table 2. The Calgary corpus is a standard benchmark for measuring compression ratios and mostly consists of English text. In particular it consists of 2 books, 5 papers, 1 bibliography, 1 collection of news articles, 3 programs, 1 terminal session, 2 object files, 1 geophysical data, and 1 bit-map b/w image. The table shows how the state of the art has improved over the years.

2 Basic Compression Algorithms

We will first discuss codes that assign a unique bit-string for coding each symbol (Huffman codes and Shannon Fano codes) and then discuss Arithmetic coding which "blends" the codes from different symbols.

2.1 Prefix Codes

- each symbol (message) is a leaf in a binary tree
- the code is encoded in the path from root to the leaf (left = 0, right = 1)
- *Property*: No code is a prefix of another code

The average length of a prefix code assuming a probability distribution on the symbols is defined as $l_a = \sum_i p_i l_i$ where p_i is the probability of the i^{th} symbol and l_i is the length of its code (the depth of its leaf).

2.2 Huffman Codes

Huffman codes are an optimal set of prefix codes. They are optimal in the sense that no other prefix code will have a shorter average length.

To build a Huffman code tree:

- start with a forest of trees each representing a symbol and with weight $w_i = p_i$
- Repeat until single tree
  - select two trees with lowest weights (w_1 and w_2)
  - combine into single tree with weight $w_1 + w_2$

Properties of Huffman Coding:

1. The length of the code for symbol i is approximately $\log_2(\frac{1}{p_i})$. (In the original lecture I said that $\lceil \log_2(\frac{1}{p_i}) \rceil$ is an upper bound on the length of each code. This is actually not true, as pointed out by a student. Consider the probabilities $.01, .30, .34, .35$. A Huffman coding will give the probability .30 a length 3, which is $> \lceil \log_2(1/.3) \rceil = 2$. It is the last time I trust "Data and Image Compression" by Gilbert Held.)
2. The average length l_a of a Huffman code matches the entropy if $p_i = \frac{1}{2^j}$ where j is any integer, for all i.

2.3 Shannon-Fano Coding

Also a prefix code, but tree is built from the top down, as opposed to being built from bottom-up as in the Huffman Coding. First sort S by probabilities, then partition list of messages into two partitions, such that the subsums of the probabilities in each partition is as close to 50% as possible. Each partition is assigned a branch with a 1 or a 0. Repeat process until tree is fully built and all messages are separated into leaves.

```
make_codes(S)
    if |S| = 1 then S[0]
    else
       sort S by probabilities
       split S into S1 and S2 such that W(S1) approx. W(S2)
       return node(make_code(S1), make_code(S2))
```

where `W(S)` represents the sum of the probabilities in the set `S`. Unlike Huffman codes, Shannon-Fano codes do not always give optimal prefix codes.

2.4 Arithmetic Coding

Given the probability of each symbol, we start with a probability range of 1. Then we separate the range of 1 into discrete partitions according to the probabilities of each symbol. So if $P(a) = 0.6$ and $P(b) = 0.4$, then 60% of the range is one partition for the symbol a. As each symbol is read from the input, the range is further refined as the lower and upper bounds of the range are modified by the probability of the new symbol. If we use L_i and U_i to denote the lower and upper bounds, and l_i and u_i to denote the range of probability for that single symbol (i.e. $l_a = 0$ and $u_a = 0.6$), then we have:

$$L_i = L_{i-1} + (U_{i-1} - L_{i-1}) \times l_i$$

and

$$U_i = L_{i-1} + (U_{i-1} - L_{i-1}) \times u_i$$

Properties:

- might not be able to emit code until seen full message (in cases where the range straddles 0.5, the first bit cannot be output until the range falls to one side of 0.5)
- asymptotically approaches entropy
- can require arbitrary precision arithmetic to implement (although we will see a nice trick in the next class)
- often used together with other methods (i.e. as a back-end)

A solution to the straddling and precision problems is to separate the input into fixed size blocks (e.g. 100 symbols), and output compressed code for each block

2.5 Deriving Probabilities

For Huffman, Shannon-Fano and Arithmetic coding we need probabilities. How do we derive them.

- Static vs. Dynamic
  - Static for a single document: We could measure the probabilities over the document we plan to send and use those. In this case we have to send the probabilities (or generated codes) to the decoder.
  - Static for a language: We could use a pre-measured set of probabilities. In this case the decoder could already know the probabilities.
  - Dynamic: We could update the probabilities as we are sending a document based on what we have seen. In this case the decoder can also calculate the probabilities on the fly as it decodes the message (we don't have to send them).
- Single symbols vs. Groups of Symbols
  - We could measure the probability of each symbol independent of its context.
  - We could measure the probability based on a context (i.e. previous symbols). This is discussed further in the next class and will usually give much better compression.

PPM: Prediction by partial matching

This is one of general-purpose algorithms that gets the best compression ratios (although it is certainly not the fastest). It is based on generating static probabilities over the current document using the symbols and their context. The algorithm can be outlined as follows.

- Try finding all multiple occurrences (matches) of strings of length n in the message (n is a preset constant).
- For leftovers, try to find multiple occurrences of strings of $n - 1$.
- repeat until all possible matches are found, and assign a probability to each string according to # of occurrences.
- Send the probabilities along with the arithmetic coding of the message that uses the probabilities.

3 Lempel-Ziv Overview

The Lempel-Ziv algorithms compress by building a table of previously seen strings. They do not use probabilities (a string is either in the table or it is not). There are two main variants based on two papers they wrote, one in 1977 (LZ77) and the other in 1978 (LZ78).

- LZ78 (Dictionary Based)

K	w	wK	output
a		a	
b	a	ab(256)	a
c	b	bc(257)	b
a	c	ca(258)	c
b	a	ab	
c	ab	abc(259)	256
a	c	ca	
-	ca	-	258

Table 3: LZ78 Encoding on "abcabca"

K	w	wK	output
a		a	
a	a	aa(256)	a
a	a	aa	
a	aa	aaa(257)	256
a	a	aa	
a	aa	aaa	
-	aaa		257

Table 4: LZ78 Encoding on "aaaaaa"

  - LZW: Lempel-Ziv-Welch
  - LZC: Lempel-Ziv Compress
  - LZMW: (Unix Compress, GIF)

- LZ77 (Sliding Window)
  - Window instead of dictionary
  - LZSS: LZ-Storer-Szymanski (gzip, ZIP, many others)

Traditionally, LZ77 is better but slower.

3.1 LZ78 and LZW

Basically starts with a string lookup table (dictionary) consisting of the standard characters, and generates a dynamic table of new string patterns of increasing length. The decompression algorithm depends on it being able to generate the same table as the compressor.

Encode:

```
w = NIL
while (K = readchar()) {
  if wK in dictionary
    w = wK
  else {
      output code for w
      add wK to dictionary
      w = K }
}
output code for w
```

Key is noting that the code for for a new string `wK` is not output until the second time it is seen. This makes it possible for the decoder to know what the code represents. If in the

input	w (at start)	output (wn)	add to dict (w wn[0])
a		a	
b	a	b	ab(256)
c	b	c	bc(257)
256	c	ab	ca(258)
258	ab	ca	abc(259)

input	w (at start)	output (wn)	add to dict (w wn[0])
a		a	
256	a	aa	aa(256)
257	aa	aaa	aaa(257)

Table 5: LZ78 Decoding on "a b c 256 258" and on "a 256 257"

`else` clause we output the new code for `wK` instead of `w` the message would be impossible to decode.

Tables 3 and 4 show two examples of the encoder. Note that the second example shows a contingency that must be accounted for, when the input has a long string of the same character. The decoder must be able to recognize this and know that code 256 is "aa". This is handled by the `else` clause below.

Decode:

```
w = lookup(readcode())
output w
while ( c = readcode()) {
  if C in dictionary
      wn = lookup(C)
  else
      wn = stringcat(w, w[0])
  output wn
  K = wn[0]
  add wK to dictionary
  w = wn
}
```

Tables 5 shows two examples of decoding.

Properties:

1. Dynamic: (e.g. works well with tar files which have many formats), adaptable.

2. No need to send the dictionary (In LZW need to send clear signals to clear dictionary/string table)

3. Can use Tries to store the dictionary/lookup tables, making lookups effectively executable in unit time.

What happens when dictionary gets too large?

1. Throw dictionary away when reaching a certain size (GIF)

2. Throw dictionary away when not effective (Unix Compress)

3. Throw Least Recently Used entry away when reaches a certain size (BTLZ - British Telecom Standard)

3.2 LZ77 (Sliding Windows)

This scheme involves a finite lookahead window, and the X most recently characters encoded as the dictionary. look in the lookahead window to find the longest string that starts at the beginning of the buffer and has already occurred in the dictionary. Then output: 1. position of prev. occurrence in window, 2. length of string, 3. next char. Then advance windows by length + 1.

Example:

```
1 2 3 4 5 6 7 8 9 10 11  12 13 14 15
a a c a a c a b c a  b || a  b  a  c <--- Data Source
<-----Dictionary-----> || <--L.B.-->
 (previously encoded)    (lookahead buffer)
 Lookahead buffer encoded as <10, 3, c>
```

LZSS Variant:
Don't output next char, instead there are two output formats:
(0, position, length)
(1, char)
Typically only output first format if length ≥ 3. Use Huffman coding to code the outputs.
Optimization: Use Trie to quickly search for match in window (gzip).
Example: Window size = 32

```
Input : aaaaaa
Output: (1, a)  (0, 31, 5)
Input : abcabca
Output: (1, a) (1, b) (1, c) (0, 29, 3) (1, a)
```